Operating Systems - Assignment

COMP2006

Jordan Yeo
17727626

Curtin University
Science and Engineering

Contents

Introduction

The following report is for the Operating Systems Assignment for 2017. It will detail how mutual exclusion was achieved for processes and threads. Also the processes and threads that accessed shared resources. Submitted alongside this report is a README, source code and test inputs and outputs.

1 Process MSSV

For this implementation of the solution, mutual exclusion is achieved by having the parent (consumer) wait for all child (producer) processes to complete execution before continuing. This waiting is implemented by having the parent wait for the semParent semaphore before accessing the shared variable, representing the number of child processes finished. Once the parent has finished creating children it will only be reading the value of the shared variable. The semParent semaphore is to prevent the critical section problem. So the child does not write to the variable as the parent reads. Once a child completes execution it waits for the semParent semaphore before acquiring a mutex lock to update a shared resource, indicating it is finished. The parent is forced to wait until the value of the shared variable shows that all children are finished.

$$wait(semParent)$$
$$\text{// critical section}$$
$$signal(semParent)$$

To ensure mutual exclusion for the shared resources (buffer1, buffer2 and counter) the child waited to acquire a mutex lock before entering its critical section to modify the shared resources. Since buffer1 was never modified and only read from this did not require a mutex lock to be obtained before reading.

$$wait(semMutex)$$
$$\text{// critical section}$$
$$signal(semMutex)$$

The semaphores required (semMutex and semParent), buffer1, buffer2 and counter were implemented using shared memory. The following POSIX shared memory functions were used to create shared memory, and the corresponding functions to close the shared memory:

$$shm\_link()$$
$$ftruncate()$$
$$mmap()$$
$$shm\_unlink()$$
$$close()$$
$$munmap()$$

Zombie processes were killed with the use of signal(SIGCHILD, SIG_IGN).

2 Thread MSSV

To achieve mutual exclusion in the multi-threaded program, the parent (consumer) must wait for all threads (producer) to complete execution. The parent uses *pthread_lock_mutex()* to lock the mutex then performs a *pthread_cond_wait()* on a condition variable. The condition variable represents how many threads are still executing.

Once a thread completes execution, it acquires a mutex lock to alter the condition variable signaling it has finished its task. The thread also performs a conditional wait in the case where the parent is using the shared variable.

$$pthread\_mutex\_lock(\&mutex)$$

$$pthread\_cond\_wait(\&use, \&mutex)$$

$$pthread\_cond\_signal(\&use);$$

$$pthread\_mutex\_unlock(\&mutex)$$

In thread MSSV the shared resources are declared as global variables. Threads have access to the variables declared global in the parent. Before a thread could access the shared resources it would first need to acquire a mutex lock. The function *pthread_lock_mutex()* blocks the caller if the mutex is in use by another. It can then alter the shared resources, counter and buffer2, before releasing the mutex lock.

To allocate the memory for buffer1, buffer2, counter and regions *malloc()* was used. The appropriate *free()* calls were used to free the allocated memory. This was done to ensure memory leaks were not present in the operation of the program.

3 Testing

3.1 Method

To test each implementation of MSSV worked as intended multiple input files were used. With these input files, multiple delays were chosen as well. Input files of various 9x9 numbers were used. Input files were used that were valid, contained one error, contained multiple errors. Also tested were smaller grids than 9x9 and empty files.

Testing was performed on the lab machines in various rooms of Building 314, Level 2.

3.2 Errors

There are no known errors in the process MSSV and the thread MSSV. Care has been taken to ensure potential memory leaks are prevented, by using the appropriate measures to free allocated memory. Memory leaks are not present in the testing of each MSSV currently performed.

3.3 Input Files

$$specTest = \begin{matrix} 6 & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & 1 \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

$$testFail = \begin{matrix} \boxed{2} & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & 1 \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

$$multiFail = \begin{matrix} \boxed{2} & 2 & 4 & 5 & 3 & 9 & 1 & 8 & 7 \\ 5 & 1 & 9 & 7 & 2 & 8 & 6 & 3 & 4 \\ 8 & 3 & 7 & 6 & 1 & 4 & 2 & 9 & 5 \\ 1 & 4 & 3 & 8 & 6 & 5 & 7 & 2 & 9 \\ 9 & 5 & 8 & 2 & 4 & 7 & 3 & 6 & \boxed{3} \\ 7 & 6 & 2 & 3 & 9 & 1 & 4 & 5 & 8 \\ 3 & 7 & 1 & 9 & 5 & 6 & 8 & 4 & 2 \\ 4 & 9 & 6 & 1 & 8 & 2 & 5 & 7 & 3 \\ 2 & 8 & 5 & 4 & 7 & 3 & 9 & 1 & 6 \end{matrix}$$

3.4 Expected Results

specTest

```
row 1 is valid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is valid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
9 out of 9 columns are valid
9 out of 9 sub-grids are valid

There are 27 valid sub-grids, and thus the solution is valid
```

testFail

```
row 1 is invalid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is valid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
8 out of 9 columns are valid
8 out of 9 sub-grids are valid

There are 24 valid sub-grids, and thus the solution is invalid
```

multiFail

```
row 1 is invalid
row 2 is valid
row 3 is valid
row 4 is valid
row 5 is invalid
row 6 is valid
row 7 is valid
row 8 is valid
row 9 is valid
7 out of 9 columns are valid
7 out of 9 sub-grids are valid

There are 21 valid sub-grids, and thus the solution is invalid
```

3.5 Actual Results

Figure 1: Processes: specTest

```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/specTest.txt 10
Validation result from process ID-19029: row 1 is valid
Validation result from process ID-19030: row 2 is valid
Validation result from process ID-19031: row 3 is valid
Validation result from process ID-19032: row 4 is valid
Validation result from process ID-19033: row 5 is valid
Validation result from process ID-19034: row 6 is valid
Validation result from process ID-19035: row 7 is valid
Validation result from process ID-19036: row 8 is valid
Validation result from process ID-19037: row 9 is valid
Validation result from process ID-19038: 9 out of 9 columns are valid
Validation result from process ID-19039: 9 out of 9 sub-grids are valid
There are 27 valid sub-grids, and thus the solution is valid
```

Figure 2: Threads: specTest

```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/specTest.txt 10
Validation result from thread ID-2979485440: row 1 is valid
Validation result from thread ID-2971092736: row 2 is valid
Validation result from thread ID-2962700032: row 3 is valid
Validation result from thread ID-2954307328: row 4 is valid
Validation result from thread ID-2945914624: row 5 is valid
Validation result from thread ID-2937521920: row 6 is valid
Validation result from thread ID-2929129216: row 7 is valid
Validation result from thread ID-2920736512: row 8 is valid
Validation result from thread ID-2912343808: row 9 is valid
Validation result from thread ID-2903951104: 9 out of 9 columns are valid
Validation result from thread ID-2895558400: 9 out of 9 sub-grids are valid
There are 27 valid sub-grids, and thus the solution is valid
```

Figure 3: Processes: testFail

```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/testFail.txt 10
Validation result from process ID-19053: row 1 is invalid
Validation result from process ID-19054: row 2 is valid
Validation result from process ID-19055: row 3 is valid
Validation result from process ID-19056: row 4 is valid
Validation result from process ID-19057: row 5 is valid
Validation result from process ID-19058: row 6 is valid
Validation result from process ID-19059: row 7 is valid
Validation result from process ID-19060: row 8 is valid
Validation result from process ID-19061: row 9 is valid
Validation result from process ID-19062: 8 out of 9 columns are valid
Validation result from process ID-19063: 8 out of 9 sub-grids are valid
There are 24 valid sub-grids, and thus the solution is invalid
```

Figure 4: Threads: testFail

```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/testFail.txt 10
Validation result from thread ID-2959283968: row 1 is invalid
Validation result from thread ID-2950891264: row 2 is valid
Validation result from thread ID-2942498560: row 3 is valid
Validation result from thread ID-2934105856: row 4 is valid
Validation result from thread ID-2925713152: row 5 is valid
Validation result from thread ID-2917320448: row 6 is valid
Validation result from thread ID-2908927744: row 7 is valid
Validation result from thread ID-2900535040: row 8 is valid
Validation result from thread ID-2892142336: row 9 is valid
Validation result from thread ID-2883749632: 8 out of 9 columns are valid
Validation result from thread ID-2875356928: 8 out of 9 sub-grids are valid
There are 24 valid sub-grids, and thus the solution is invalid
```

Figure 5: Processes: multiFail

```
[17727626@lab221-c01 partA]$ ./mssv ../testFiles/multiFail.txt 10
Validation result from process ID-19070: row 1 is invalid
Validation result from process ID-19071: row 2 is valid
Validation result from process ID-19072: row 3 is valid
Validation result from process ID-19073: row 4 is valid
Validation result from process ID-19074: row 5 is invalid
Validation result from process ID-19075: row 6 is valid
Validation result from process ID-19076: row 7 is valid
Validation result from process ID-19077: row 8 is valid
Validation result from process ID-19078: row 9 is valid
Validation result from process ID-19079: 7 out of 9 columns are valid
Validation result from process ID-19080: 7 out of 9 sub-grids are valid
There are 21 valid sub-grids, and thus the solution is invalid
```

Figure 6: Threads: multiFail

```
[17727626@lab221-b01 partB]$ ./mssv ../testFiles/multiFail.txt 10
Validation result from thread ID-3998418688: row 1 is invalid
Validation result from thread ID-3990025984: row 2 is valid
Validation result from thread ID-3981633280: row 3 is valid
Validation result from thread ID-3973240576: row 4 is valid
Validation result from thread ID-3964847872: row 5 is invalid
Validation result from thread ID-3956455168: row 6 is valid
Validation result from thread ID-3990025984: row 7 is valid
Validation result from thread ID-3813844736: row 8 is valid
Validation result from thread ID-3948062464: row 9 is valid
Validation result from thread ID-3939669760: 7 out of 9 columns are valid
Validation result from thread ID-3931277056: 7 out of 9 sub-grids are valid
There are 21 valid sub-grids, and thus the solution is invalid
```

4 README

4.1 Purpose

The program validates an input file containing a sudoku solution. There are two versions. One utilising processes and the other utilising threads.

4.2 Running the Program

To compile the C files into an executable format.

```
make
```

To run the program there are two options.

Option 1:

```
make run
```

This will only let you run with the preset parameters and they will need to be altered in the Makefile to test other *input files* and *delays*

```
INPUT: ../testFiles/specTest.txt
DELAY: 10
```

Option 2:

```
./mssv <inputFile> <delay>
```

Between each time the program is run, the following command should be entered and executed. This is to delete the logfile produced from an invalid test file.

```
make clean
```

4.3 Files

partA

- Makefile
- mssv.c
- mssv.h

partB

- Makefile
- mssv.c
- mssv.h

testFiles

- test files

References

[1] *Interprocess communication using POSIX Shared Memory in Linux*. URL: https://www.softprayog.in/programming/interprocess-communication-using-posix-shared-memory-in-linux.

[2] *POSIX Semaphores*. URL: https://www.softprayog.in/programming/posix-semaphores.

[3] *POSIX Threads Programming in C*. URL: https://www.softprayog.in/programming/posix-threads-programming-in-c.

[4] *POSIX Threads Synchronization in C*. URL: https://www.softprayog.in/programming/posix-threads-synchronization-in-c.

5 Appendices

5.1 Processes: mssv.c

```
#include "mssv.h"

int main (int argc, char* argv[])
{
    // Validate command line parameters
    validateUse(argc, argv);

    // Rename command line parameters
    char* inputFile = argv[1];
    int maxDelay = atoi(argv[2]);

    // Variables
    int i, pid, processNum, numbers[] = {0,0,0,0,0,0,0,0,0};
    Region* region;
    sem_t semMutex, semParent, *semaphores;

    // File Descriptors
    int buff1FD, buff2FD, counterFD, semFD, regionFD, resFD;

    // Shared memory pointers
    int *buff2Ptr, *countPtr, *resourceCount, (*buff1Ptr)[NINE][NINE];

    // Create shared memory
    initMemory( &buff1FD, &buff2FD, &counterFD, &semFD, &regionFD, &resFD);

    // Map shared memory to pointers
    mapMemory(&buff1FD, &buff2FD, &counterFD, &semFD, &regionFD, &resFD,
                &buff1Ptr, &buff2Ptr, &countPtr, &semaphores, &region,
                    &resourceCount);

    // Initialise semaphores
    if ((sem_init(&semMutex, 1, 1)== 1) ||(sem_init(&semParent, 1, 1) == 1))
    {
        fprintf(stderr, "Could not initialise semaphores\n");
        exit(1);
    }

    semaphores[0] = semMutex;
    semaphores[1] = semParent;

    // Initialise parameters
    *countPtr = 0;
    pid = -1;
    processNum = 0;

    // Read input file
    readFile(inputFile, NINE, NINE, buff1Ptr);

    // Parent aquires lock of resourceCount
    sem_wait(&(semaphores[1])); // Lock child

    *resourceCount = 0;

    // Create child processes for
    while( processNum < NUMPROCESSES && pid != 0 )
    {
        signal(SIGCHLD, SIG_IGN); // Kill zombie processNum-1
        pid = fork();

        // Allow the parent to increment shared variable count
        if ( pid > 0)
        {
            *resourceCount = *resourceCount + 1;
```

```
64          }
65          processNum++;
66      }
67
68      if( pid == 0) // Child process
69      {
70          childManager(region, semaphores, buff1Ptr, buff2Ptr, countPtr,
71                                  resourceCount,  processNum, numbers, maxDelay );
72      }
73      else if ( pid > 0) // Parent process
74      {
75          parentManager(region, semaphores, countPtr, resourceCount);
76
77          // Clean up shared memory
78          cleanMemory(&buff1Ptr, &buff2Ptr, &countPtr, &semaphores,
79                          &region, &resourceCount, buff1FD, buff2FD, counterFD,
80                                semFD, regionFD, resFD);
81      }
82      else // Unsuccessful child process creation attempt
83      {
84          fprintf(stderr, "Unable to create child processes. Please run \"killall mssv\"\n");
85      }
86  }
87
88  /*****************************************************************************/
89
90  /**
91   * Read the contents of the input file passed as a command line argument
92   * @param inputFile File to be read
93   * @param rows      Number of rows in matrix
94   * @param cols      Number of columns in matrix
95   * @param buffer    Matrix to store contents of input file
96   */
97  void readFile(char* inputFile, int rows, int cols, int (*buffer)[rows][cols])
98  {
99      FILE* inStrm;
100     int i, j;
101
102     inStrm = fopen(inputFile, "r"); // Open file for reading
103
104     if (inStrm == NULL) // Check file opened correctly
105     {
106         perror("Error opening file for reading\n");
107         exit(1);
108     }
109
110     // Store contents of file in 2D array
111     for( i = 0; i < rows; i++ )
112     {
113         for ( j = 0; j < cols; j++ )
114         {
115             fscanf( inStrm, "%d", &(*(buffer))[i][j] );
116         }
117     }
118
119     fclose(inStrm); // Close file
120 }
121
122 /**
123  * Write the invalid regions to log file
124  * @param region Sub region
125  * @param format String to be written
126  */
127 void writeFile(Region* region, char* format)
128 {
129     char* filename = "logfile";
130     FILE* outFile;
131     int val;
```

```
132
133     outFile = fopen(filename, "a"); // Open file for appending
134     if (outFile == NULL) // Check file opened correctly
135     {
136         perror("Error opening file for writing\n");
137         exit(1);
138     }
139
140     fprintf(outFile, "process ID-%d: %s",region->pid, format);
141
142     fclose(outFile); // Close file
143 }
144
145 /**
146  * Set each index to zero
147  * @param numbers Array to be reset
148  */
149 void resetArray(int numbers[])
150 {
151     for (int i = 0; i < NINE; i++)
152     {
153         numbers[i] = 0;
154     }
155 }
156
157 /**
158  * Check if the contents of the array has any value other than one
159  * @param  numbers Array to be checked
160  * @return         Status of array being valid or not
161  */
162 int checkValid(int numbers[])
163 {
164     for (int j = 0; j < NINE; j++)
165     {
166         if ( numbers[j] != 1)
167         {
168             return FALSE;
169         }
170     }
171
172     return TRUE;
173 }
174
175 /**
176  * Handles the routine for the parent process. Outputs the result to the screen
177  * @param region        Array containing each region struct
178  * @param semaphores    Array of all semaphores
179  * @param countPtr      Pointer to shared memory counter
180  * @param resourceCount Status of number of child processes executing
181  */
182 void parentManager(Region *region, sem_t *semaphores, int* countPtr,
183                         int* resourceCount)
184 {
185     char *type, *message;
186     sem_post(&(semaphores[1])); // Unlock child
187     int done = FALSE;
188     int position;
189
190     while( !done ) // Wait for all children to finish executing
191     {
192         //printf("Parent Waiting for Children\n");
193         sem_wait(&(semaphores[1])); // Lock child
194         if ( *resourceCount == 0)
195         {
196             done = TRUE;
197         }
198         sem_post(&(semaphores[1])); // Unlock child
199
```

```
200     }
201
202     for(int ii = 0; ii < NUMPROCESSES; ii++)
203     {
204         sem_wait(&(semaphores[0])); //Lock mutex
205         if (region[ii].type == ROW)
206         {
207             position = region[ii].positionX;
208             type = "invalid";
209
210             if (region[ii].valid == TRUE)
211             {
212                 type = "valid";
213             }
214             printf("Validation result from process ID-%d: row %d is %s\n",
215                                     region[ii].pid, position, type);
216         }
217         else if (region[ii].type == COL)
218         {
219             type = "column";
220             position = region[ii].positionX;
221             printf("Validation result from process ID-%d: %d out of 9 columns are valid\n"
222                         ,region[ii].pid, region[ii].positionX);
223         }
224         else
225         {
226             type = "sub-grid";
227             position = region[ii].positionX;
228
229             printf("Validation result from process ID-%d: %d out of 9 sub-grids are valid\n"
230                         ,region[ii].pid, region[ii].positionX);
231
232         }
233
234         sem_post(&(semaphores[0])); //Unlock mutex
235     }
236
237
238   if (*countPtr == 27)
239   {
240       message = "valid";
241   }
242   else
243   {
244       message = "invalid";
245   }
246
247   printf("There are %d valid sub-grids, and thus the solution is %s\n", *countPtr, message);
248 }
249
250
251 /**
252  * Routine for child processes. Check the validity of sub region.
253  * @param region         Sub-region struct for each process
254  * @param semaphores     Array of semaphores
255  * @param buff1Ptr       Pointer to buffer1 in shared memory
256  * @param buff2Ptr       Pointer to buffer2 in shared memory
257  * @param countPtr       Pointer to counter in shared memory
258  * @param resourceCount  Pointer to resourceCount in shared memory
259  * @param processNum     Child process number
260  * @param numbers        Array of numbers to check validity of sub region
261  * @param maxDelay       Delay for each process
262  */
263 void childManager(Region *region, sem_t *semaphores, int (*buff1Ptr)[NINE][NINE],
264                     int *buff2Ptr, int* countPtr, int* resourceCount,
265                         int processNum, int *numbers, int maxDelay )
266 {
267     char format[500];
```

```
268     int numValid, comma = 0;
269
270     // Generate random maxDelay
271     srand((unsigned) getpid() );
272
273     maxDelay = ( rand() % maxDelay ) + 1;
274
275     if( processNum <= 9) // Check a row in buffer1
276     {
277         for (int i = 0; i < NINE; i++)
278         {
279             // Update numbers array
280             numbers[((*buff1Ptr)[processNum-1][i])-1]++;
281         }
282
283         sleep(maxDelay); // Sleep
284         sem_wait(&(semaphores[0])); //Lock mutex
285
286         // Update region struct
287         region[processNum-1].type = ROW;
288         region[processNum-1].positionX = processNum;
289         region[processNum-1].pid = getpid();
290         region[processNum-1].valid = checkValid(numbers);
291
292         numValid = 0;
293         if (region[processNum-1].valid == TRUE)
294         {
295             numValid = 1;
296         }
297         else // Write to log file
298         {
299             sprintf(format, "row %d is invalid\n", processNum);
300             writeFile(&(region[processNum-1]), format);
301         }
302
303         buff2Ptr[processNum-1] = numValid; // Update buffer2
304
305         *countPtr = *countPtr + numValid; // Update counter
306
307         sem_post(&(semaphores[0])); // Unlock child
308
309     }
310     else if(processNum == 10) // Check all columns
311     {
312         sprintf(format, "column ");
313
314         int validCol = 0;
315         for ( int nn = 0; nn < NINE; nn++) // Iterate through each column
316         {
317             for(int ii = 0; ii < NINE; ii++) // Iterate through each row
318             {
319                 numbers[(*buff1Ptr)[ii][nn]-1]++; // Update numbers array
320             }
321
322             if ( checkValid( numbers) == TRUE )
323             {
324               validCol++;
325             }
326             else
327             {
328                 if (comma == 0)
329                 {
330                     comma = 1;
331                     sprintf(format + strlen(format), "%d", nn+1);
332                 }
333                 else
334                 {
335                     sprintf(format + strlen(format), ", %d ", nn+1);
```

```
336                 }
337             }
338 
339             resetArray(numbers);
340         }
341 
342         sleep(maxDelay);
343         if (validCol == 8)
344         {
345             sprintf(format + strlen(format), " is invalid\n");
346         }
347         else
348         {
349             sprintf(format + strlen(format), "are invalid\n");
350         }
351         sem_wait(&(semaphores[0])); //Lock mutex
352 
353         // Update region struct
354         region[processNum-1].type = COL;
355         region[processNum-1].positionX = validCol;
356         region[processNum-1].pid = getpid();
357         if(validCol != 9)
358         {
359             writeFile(&(region[processNum-1]), format);
360         }
361 
362         numValid = region[processNum-1].positionX;
363 
364         buff2Ptr[processNum-1] = validCol; // Update buffer2
365 
366         *countPtr = *countPtr + validCol; // Update counter
367 
368         sem_post(&(semaphores[0])); // Unlock mutex
369     }
370     else if( processNum == 11) // Check sub-grids
371     {
372         sprintf(format, "sub-grid ");
373 
374         int validSub = 0;
375 
376         // Iterate through each of the 9 3x3 sub-grid
377         for ( int jj = 0; jj < 3; jj++)
378         {
379             for (int kk = 0; kk < 3; kk++)
380             {
381                 for (int ll = jj*3; ll < jj*3+3; ll++)
382                 {
383                     for (int mm = kk*3; mm < kk*3+3; mm++)
384                     {
385                         // Update numbers array
386                         numbers[(*buff1Ptr)[ll][mm]-1]++;
387                     }
388               }
389 
390               if ( checkValid(numbers) == TRUE )
391               {
392                     validSub++;
393               }
394               else // Update string for log file
395               {
396                   if (comma == 0)
397                   {
398                       comma = 1;
399                       sprintf(format+strlen(format), "[%d..%d, %d..%d]",
400                                   jj*3+1, jj*3+3, kk*3+1, kk*3+3);
401                   }
402                   else
403                   {
```

```
404                         sprintf(format+strlen(format), ", [%d..%d, %d..%d] ",
405                                 jj*3+1, jj*3+3, kk*3+1, kk*3+3);
406                     }
407                 }
408                 resetArray(numbers);
409             }
410 
411         }
412 
413         sleep(maxDelay);
414         if (validSub == 8)
415         {
416             sprintf(format+strlen(format), " is invalid\n");
417         }
418         else
419         {
420             sprintf(format+strlen(format), "are invalid\n");
421         }
422         sem_wait(&(semaphores[0])); //Lock mutex
423 
424         // Update region struct
425         region[processNum-1].type = SUB_REGION;
426         region[processNum-1].positionX = validSub;
427         region[processNum-1].pid = getpid();
428 
429         if(validSub != 9) // Write to log file
430         {
431             writeFile(&(region[processNum-1]), format);
432         }
433 
434         buff2Ptr[processNum-1] = validSub; // Update buffer 2
435 
436         *countPtr = *countPtr + validSub; // Update counter
437 
438         sem_post(&(semaphores[0])); // Unlock mutex
439     }
440 
441     // Child signals it is finished by incremented resourceCount
442     sem_wait(&(semaphores[1])); // Lock child
443         *resourceCount = *resourceCount - 1;
444     sem_post(&(semaphores[1])); // Unlock child
445 
446 }
447 
448 /**
449  * Initalise shared memory constructs
450  * @param buff1FD   File descriptor for buffer1
451  * @param buff2FD   File descriptor for buffer2
452  * @param counterFD File descriptor for counter
453  * @param semFD     File descriptor for semaphores
454  * @param regionFD  File descriptor for regions
455  * @param resFD     File descriptor for resourceCount
456  */
457 void initMemory( int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
458                  int* regionFD, int* resFD)
459 {
460 
461     // Create shared memory
462     *buff1FD = shm_open("buffer1", O_CREAT | O_RDWR, 0666);
463     *buff2FD = shm_open("buffer2", O_CREAT | O_RDWR, 0666);
464     *counterFD = shm_open("counter", O_CREAT | O_RDWR, 0666);
465     *semFD = shm_open("semaphores", O_CREAT | O_RDWR, 0666);
466     *regionFD = shm_open("region", O_CREAT | O_RDWR, 0666);
467     *resFD = shm_open("resources", O_CREAT | O_RDWR, 0666);
468 
469     // Check shared memory was created correctly
470     if ( *buff1FD == -1 || *buff2FD == -1 || *counterFD == -1 || *semFD == -1 ||
471             *regionFD == -1 || *resFD == -1 )
```

```
472     {
473         fprintf( stderr, "Error creating shared memory blocks\n" );
474         exit(1);
475     }
476 
477     // Set size of shared memory constructs
478     if ( ftruncate(*buff1FD, sizeof(int) * NINE * NINE) == -1 )
479     {
480         fprintf( stderr, "Error setting size of buffer1" );
481         exit(1);
482     }
483 
484     if ( ftruncate(*buff2FD, sizeof(int) * NUMPROCESSES) == -1 )
485     {
486         fprintf( stderr, "Error setting size of buffer2" );
487         exit(1);
488     }
489 
490     if ( ftruncate(*counterFD, sizeof(int)) == -1 )
491     {
492         fprintf( stderr, "Error setting size of counter" );
493         exit(1);
494     }
495 
496     if ( ftruncate(*semFD, sizeof(sem_t) * 2 ) == -1 )
497     {
498         fprintf( stderr, "Error setting size of semaphores" );
499         exit(1);
500     }
501 
502     if ( ftruncate(*regionFD, sizeof(Region)*NUMPROCESSES) == -1 )
503     {
504         fprintf( stderr, "Error setting size of regions" );
505         exit(1);
506     }
507 
508     if ( ftruncate(*resFD, sizeof(int)) == -1 )
509     {
510         fprintf( stderr, "Error setting size of resourceCount" );
511         exit(1);
512     }
513 }
514 
515 /**
516  * Map shared memory to addresses
517  * @param buff1FD       File descriptor for buffer1
518  * @param buff2FD       File descriptor for buffer2
519  * @param counterFD     File descriptor for counter
520  * @param semFD         File descriptor for semaphores
521  * @param regionFD      File descriptor for regions
522  * @param resFD         File descriptor for resourceCount
523  * @param buff1Ptr      Pointer to buffer1 in shared memory
524  * @param buff2Ptr      Pointer to buffer2 in shared memory
525  * @param countPtr      Pointer to counter in shared memory
526  * @param semaphores    Array of semaphores
527  * @param region        Array of region structs
528  * @param resourceCount Pointer to resourceCount in shared memory
529  */
530 void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
531                    int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE], int (**buff2Ptr),
532                         int** countPtr, sem_t** semaphores, Region** region, int**
    resourceCount)
533 {
534     // Memory mapping
535     *buff2Ptr = (int*) mmap(NULL, sizeof(int)*NINE*NINE,
536                 PROT_READ | PROT_WRITE, MAP_SHARED, *buff2FD, 0);
537     *buff1Ptr = mmap(NULL, sizeof(int)*NUMPROCESSES,
538                 PROT_READ | PROT_WRITE, MAP_SHARED, *buff1FD, 0);
```

```
539     *countPtr = (int*) mmap(NULL, sizeof(int),
540                 PROT_READ | PROT_WRITE, MAP_SHARED, *counterFD, 0);
541     *semaphores = mmap(NULL, sizeof(sem_t) * 2,
542                 PROT_READ | PROT_WRITE, MAP_SHARED, *semFD, 0);
543     *region = mmap(NULL, sizeof(Region)*NUMPROCESSES,
544                 PROT_READ | PROT_WRITE, MAP_SHARED, *regionFD, 0);
545     *resourceCount = mmap(NULL, sizeof(int),
546                 PROT_READ | PROT_WRITE, MAP_SHARED, *resFD, 0);
547 }
548
549 /**
550  * Validate command line parameters
551  * @param argc number of parameters
552  * @param argv command line parameters
553  */
554 void validateUse(int argc, char* argv[])
555 {
556     // Ensure correct number of command line parameters
557     if (argc != 3)
558     {
559         printf("Ensure there are the correct number of parameters\n");
560         exit(1);
561     }
562
563     // Ensure maxDelay is positive
564     if ( atoi(argv[2]) < 0)
565     {
566         printf("The maxDelay must be non-negative\n");
567         exit(1);
568     }
569 }
570
571 /**
572  * Close and destroy, semaphores and shared memory constructs
573  * @param buff1Ptr      Pointer to buffer1 in shared memory
574  * @param buff2Ptr      Pointer to buffer2 in shared memory
575  * @param countPtr      Pointer to counter in shared memory
576  * @param semaphores    Array of semaphores
577  * @param region        Array of region structs
578  * @param resourceCount Pointer to resourceCount in shared memory
579  * @param buff1FD       File descriptor for buffer1
580  * @param buff2FD       File descriptor for buffer2
581  * @param counterFD     File descriptor for counter
582  * @param semFD         File descriptor for semaphores
583  * @param regionFD      File descriptor for regions
584  * @param resFD         File descriptor for resourceCount
585  */
586 void cleanMemory(int (**buff1Ptr)[NINE][NINE], int **buff2Ptr, int** countPtr,
587                          sem_t **semaphores, Region **region,
588                             int** resourceCount, int buff1FD, int buff2FD,
589                                 int counterFD, int semFD, int regionFD,
590                                     int resFD )
591 {
592     // Close semaphores
593     sem_close(&((*semaphores)[0]));
594     sem_close(&((*semaphores)[1]));
595
596     // Destroy semaphores
597     sem_destroy(&((*semaphores)[0]));
598     sem_destroy(&((*semaphores)[1]));
599
600     // Clean up shared memory
601     shm_unlink("buffer1");
602     shm_unlink("buffer2");
603     shm_unlink("counter");
604     shm_unlink("semaphores");
605     shm_unlink("region");
606     shm_unlink("resources");
```

```
607
608     // Close file descriptors
609     close(buff1FD);
610     close(buff2FD);
611     close(counterFD);
612     close(semFD);
613     close(regionFD);
614     close(resFD);
615
616     // Unmap memory
617     munmap(*buff1Ptr, sizeof(int)*NINE*NINE);
618     munmap(*buff2Ptr, sizeof(int)*NUMPROCESSES);
619     munmap(*countPtr, sizeof(int));
620     munmap(*semaphores, sizeof(sem_t)*2);
621     munmap(*region, sizeof(Region)*NUMPROCESSES);
622     munmap(*resourceCount, sizeof(int));
623
624     // Unlink shared memory constructs
625     shm_unlink("buffer1");
626     shm_unlink("buffer2");
627     shm_unlink("counter");
628     shm_unlink("semaphores");
629     shm_unlink("region");
630     shm_unlink("resources");
631
632 }
```

5.2 Processes: mssv.h

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <time.h>

#define NINE 9
#define SUB 3
#define NUMPROCESSES 11
#define FALSE 0
#define TRUE !FALSE

typedef enum {ROW, COL, SUB_REGION} Region_Type;

typedef struct
{
    Region_Type type;
    int positionX;
    pid_t pid;
    int valid;
} Region;


void readFile(char* inputFile, int rows, int cols, int (*buffer)[rows][cols]);
void writeFile(Region* region, char* format);
void resetArray(int numbers[]);
int checkValid(int numbers[]);
void parentManager(Region *region, sem_t *semaphores, int* countPtr,
                        int* resourceCount );
void childManager(Region *region, sem_t *semaphores,
                    int (*buff1Ptr)[NINE][NINE], int *buff2Ptr, int* countPtr,
                        int* resourceCount, int processNum, int *numbers,
                            int maxDelay );
void initMemory( int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD);
void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE],
                        int (**buff2Ptr), int** countPtr, sem_t** semaphores,
                            Region** region, int** resourceCount);
void validateUse(int argc, char* argv[]);
void cleanMemory(int (**buff1Ptr)[NINE][NINE], int **buff2Ptr, int** countPtr,
                    sem_t **semaphores, Region **region, int** resourceCount,
                         int buff1FD, int buff2FD, int counterFD, int semFD,
                            int regionFD, int resFD);
```

5.3 Processes: Makefile

```
# Makefile For Sudoku Solution Validator
# COMP2006 Assignment
# Last Modified: 12/04/17
# Jordan Yeo - 17727626

# MAKE VARIABLES
EXEC1 = mssv
OBJ1 = mssv.o
CFLAGS = -std=c99 -pthread -D _XOPEN_SOURCE=500 -lrt
CC = gcc
INPUT = ../testFiles/specTest.txt
DELAY = 1

# RULES + DEPENDENCIES
$(EXEC1) : $(OBJ1)
  $(CC) $(OBJ1) -o $(EXEC1) $(CFLAGS)

mssv.o : mssv.c mssv.h #fileIO.h
  $(CC) -c mssv.c $(CFLAGS)

clean:
  rm -f $(EXEC1) $(OBJ1) logfile

run:
  ./$(EXEC1) $(INPUT) $(DELAY)
```

5.4 Threads: mssv.c

```c
#include "mssv.h"

pthread_mutex_t mutex; // Mutex
pthread_cond_t use;  // condition for if the global variable is in use

int **buff1, *buff2, *counter, maxDelay, inUse;
Region *regions;

int main (int argc, char* argv[])
{
    // Validate command line parameters
    validateUse(argc, argv);

    // Rename command line parameters
    char* inputFile = argv[1];
    maxDelay = atoi(argv[2]);

    // Variables
    pthread_t threads[11];

    // Allocate  memory
    initMemory( &buff1, &buff2, &counter, &regions);

    *counter = 0;
    // Read input file
    readFile(inputFile, NINE, NINE, &buff1);

    // Initialise mutex and condition
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&use, NULL);
    inUse = 0;

    // Create threads
    for(int i = 0; i < NUMTHREADS; i++)
    {
        if ( i < NINE) // Initialise region struct for row threads
        {
            regions[i].type = ROW;
        }
        else if( i == NINE) // Initialise region struct for columns thread
        {
            regions[i].type = COL;
        }
        else // Initialise region struct for sub-grids thread
        {
            regions[i].type = SUB_GRID;
        }

        regions[i].position = i;
        resetArray(regions[i].numbers);
        // Create thread
        pthread_create(&(threads[i]), NULL, childManager, &(regions[i]));
        inUse++;
    }

    parentManager(); // Parent logic

    cleanMemory(); // Clean up malloc'd memory

}

/*****************************************************************************/

/**
 * Initalise memory constructs
 * @param buff1    buffer1 2D array
```

```
67  * @param buff2    buffer2 1D array
68  * @param counter  counter variable
69  * @param regions  Region struct 1D array
70  */
71 void initMemory(int*** buff1, int** buff2, int** counter, Region** regions)
72 {
73     // Initialise
74     *buff1 = (int**) malloc(sizeof(int*)* NINE);
75     for (int i = 0; i < NINE; i++)
76     {
77         (*buff1)[i] = (int*) malloc(sizeof(int)* NINE);
78     }
79     *buff2 = (int*) malloc(sizeof(int)* NUMTHREADS);
80     *counter = (int*) malloc(sizeof(int));
81     *regions = (Region*) malloc(sizeof(Region)* NUMTHREADS);
82 }
83
84
85 /**
86  * Free malloc'd memory and destroy mutex and conditions
87  */
88 void cleanMemory()
89 {
90     pthread_mutex_destroy(&mutex);
91     pthread_cond_destroy(&use);
92     for (int i = 0; i < NINE; i++)
93     {
94         free(buff1[i]);
95     }
96     free(buff1);
97     free(buff2);
98     free(counter);
99     free(regions);
100 }
101
102
103 /**
104  * Read the contents of the input file passed as a command line argument
105  * @param inputFile File to be read
106  * @param rows      Number of rows in matrix
107  * @param cols      Number of columns in matrix
108  * @param buffer    Matrix to store contents of input file
109  */
110 void readFile(char* inputFile, int rows, int cols, int***buffer )
111 {
112     FILE* inStrm;
113     int i, j;
114
115     inStrm = fopen(inputFile, "r"); // Open file for reading
116
117     if (inStrm == NULL) // Check file opened correctly
118     {
119         perror("Error opening file for reading\n");
120         exit(1);
121     }
122
123     // Store contents of file in 2D array
124     for( i = 0; i < rows; i++ )
125     {
126         for ( j = 0; j < cols; j++ )
127         {
128             fscanf( inStrm, "%d", &(*(buffer))[i][j] );
129         }
130     }
131     fclose(inStrm); // Close file
132 }
133
134 /**
```

```
135  * Write the invalid regions to log file
136  * @param region Sub region
137  * @param format String to be written
138  */
139 void writeFile(Region* region, char* format)
140 {
141     char* filename = "logfile";
142     FILE* outFile;
143     int val;
144 
145     outFile = fopen(filename, "a"); // Open file for appending
146     if (outFile == NULL) // Check file opened correctly
147     {
148         perror("Error opening file for writing\n");
149         exit(1);
150     }
151 
152     fprintf(outFile, "thread ID-%d: %s",region->tid, format);
153 
154     fclose(outFile); // Close file
155 }
156 
157 /**
158  * Set each index to zero
159  * @param numbers Array to be reset
160  */
161 void resetArray(int numbers[])
162 {
163     for (int i = 0; i < NINE; i++)
164     {
165         numbers[i] = 0;
166     }
167 }
168 
169 /**
170  * Check if the contents of the array has any value other than one
171  * @param  numbers Array to be checked
172  * @return         Status of array being valid or not
173  */
174 int checkValid(int numbers[])
175 {
176     for (int j = 0; j < NINE; j++)
177     {
178         if ( numbers[j] != 1)
179         {
180             return FALSE;
181         }
182     }
183 
184     return TRUE;
185 }
186 
187 /**
188  * Handles the routine for the parent. Outputs the result to the screen
189  * @param threads ID of child threads
190  */
191 void parentManager()
192 {
193     char *type, *message;
194     int done = FALSE;
195     int position;
196 
197     pthread_mutex_lock(&mutex); // Lock mutex
198     while ( inUse > 0 ) // Wait while children are executing
199     {
200         pthread_cond_wait(&use, &mutex);
201     }
202 
```

```
203     pthread_cond_signal(&use);
204     pthread_mutex_unlock(&mutex); // Unlock mutex
205 
206     for(int ii = 0; ii < NUMTHREADS; ii++)
207     {
208         pthread_mutex_lock(&mutex); // Lock mutex
209 
210         if (regions[ii].type == ROW)
211         {
212             type = "row";
213             position = regions[ii].position;
214             if ( regions[ii].valid == TRUE)
215             {
216                 printf("Validation result from thread ID-%u: %s %d is valid\n",
217                                         regions[ii].tid,type, position+1);
218             }
219             else
220             {
221                 printf("Validation result from thread ID-%u: %s %d is invalid\n",
222                           regions[ii].tid, type, position+1);
223             }
224         }
225         else if (regions[ii].type == COL)
226         {
227             type = "column";
228             printf("Validation result from thread ID-%u: %d out of 9 columns are valid\n"
229                     ,regions[ii].tid, regions[ii].count);
230         }
231         else
232         {
233             type = "sub-grid";
234             printf("Validation result from thread ID-%u: %d out of 9 sub-grids are valid\n"
235                     ,regions[ii].tid, regions[ii].count);
236         }
237 
238         pthread_mutex_unlock(&mutex);
239     }
240 
241 
242   if (*counter == 27)
243   {
244       message = "valid";
245   }
246   else
247   {
248       message = "invalid";
249   }
250 
251   printf("There are %d valid sub-grids, and thus the solution is %s\n", *counter, message);
252 
253 }
254 
255 /**
256  * Routine for child threads. Check the validity of sub region.
257  * @param args Void pointer to Region struct for the child
258  */
259 void* childManager(void* args )
260 {
261     char format[500];
262     int numValid;
263     Region* region = ((Region*)(args));
264     int threadNum = region->position;
265     int comma = 0;
266     int delay;
267 
268     // Generate random maxDelay
269     srand((unsigned) pthread_self() );
270     delay = ( rand() % delay ) + 1;
```

```
271
272     if( region->type == ROW ) // Check row in buffer1
273     {
274
275         // Check rows
276         for (int i = 0; i < NINE; i++)
277         {
278             // Update numbers array
279             region->numbers[((buff1)[threadNum][i])-1]++;
280         }
281
282         sleep(delay); // Sleep
283         pthread_mutex_lock(&mutex); // Lock mutex
284
285         // Update region struct
286         region->tid = pthread_self();
287         region->valid = checkValid(region->numbers);
288
289         // Update buffer2
290         numValid = 0;
291         if (region->valid == TRUE)
292         {
293             numValid = 1;
294             region->count = numValid;
295         }
296         else // Write to log file
297         {
298             region->count = numValid;
299             sprintf(format, "row %d is invalid\n", threadNum+1);
300             writeFile((region), format);
301         }
302
303         buff2[threadNum] = numValid; // Update buffer2
304
305         *counter = *counter + numValid; // Update counter
306
307         pthread_mutex_unlock(&mutex); // Unlock mutex
308
309     }
310     else if( region->type == COL ) // Check all columns
311     {
312         sprintf(format, "column ");
313         int validCol = 0;
314         for ( int nn = 0; nn < NINE; nn++) // Iterate through each column
315         {
316             for(int ii = 0; ii < NINE; ii++) // Iterate through each row
317             {
318                 // Update numbers array
319                 region->numbers[((buff1)[ii][nn])-1]++;
320             }
321
322             // Check if the column is valid
323             if ( checkValid( region->numbers) == TRUE )
324             {
325               validCol++;
326             }
327             else
328             {
329                 if (comma == 0)
330                 {
331                     comma = 1;
332                     sprintf(format + strlen(format), "%d", nn+1);
333                 }
334                 else
335                 {
336                     sprintf(format + strlen(format), ", %d ", nn+1);
337                 }
338             }
```

```
339             resetArray(region->numbers);
340         }
341 
342         sleep(delay);
343         if (validCol == 8)
344         {
345             sprintf(format + strlen(format), " is invalid\n");
346         }
347         else
348         {
349             sprintf(format + strlen(format), "are invalid\n");
350         }
351         pthread_mutex_lock(&mutex); // Lock mutex
352 
353         // Update region struct
354         region->count = validCol;
355         region->tid = pthread_self();
356         if(validCol != 9)
357         {
358             writeFile((region), format);
359         }
360 
361         numValid = region->count;
362 
363         buff2[threadNum] = validCol; // Update buffer2
364 
365         *counter = *counter + validCol; // Update counter
366 
367         pthread_mutex_unlock(&mutex); // Unlock mutex
368     }
369     else if( region->type == SUB_GRID ) // Check all sub-grids
370     {
371         sprintf(format, "sub-grid ");
372 
373         int validSub = 0;
374 
375         // Iterate through each of the 9 3x3 sub-grid
376         for ( int jj = 0; jj < 3; jj++)
377         {
378             for (int kk = 0; kk < 3; kk++)
379             {
380                 for (int ll = jj*3; ll < jj*3+3; ll++)
381                 {
382                     for (int mm = kk*3; mm < kk*3+3; mm++)
383                     {
384                         // Update numbers array
385                         region->numbers[((buff1)[ll][mm])-1]++;
386                     }
387                 }
388 
389                 if ( checkValid(region->numbers) == TRUE )
390                 {
391                     validSub++;
392                 }
393                 else // Update string for log file
394                 {
395                     if (comma == 0)
396                     {
397                         comma = 1;
398                         sprintf(format+strlen(format), "[%d..%d, %d..%d]",
399                                 jj*3+1, jj*3+3, kk*3+1, kk*3+3);
400                     }
401                     else
402                     {
403                         sprintf(format+strlen(format), ", [%d..%d, %d..%d]",
404                                 jj*3+1, jj*3+3, kk*3+1, kk*3+3);
405                     }
406                 }
```

```
407                 resetArray(region->numbers);
408             }
409 
410         }
411 
412         sleep(delay);
413         if( validSub == 8)
414         {
415             sprintf(format+strlen(format), " is invalid\n");
416         }
417         else
418         {
419             sprintf(format+strlen(format), " are invalid\n");
420         }
421         pthread_mutex_lock(&mutex); // Lock mutex
422 
423         // Update region struct
424         region->count= validSub;
425         region->tid = pthread_self();
426 
427         if(validSub != 9)
428         {
429             writeFile((region), format);
430         }
431 
432         buff2[threadNum] = validSub; // Update buffer2
433 
434         *counter = *counter + validSub; // Update counter
435 
436         pthread_mutex_unlock(&mutex); // Unlock mutex
437 
438     }
439 
440     // Child signals it is finished by incremented resourceCount
441     pthread_mutex_lock(&mutex); // Lock mutex
442 
443     while( inUse == 0)
444     {
445         pthread_cond_wait(&use, &mutex);
446     }
447     inUse--; // Decrease count of child processes running
448     if (inUse == 0)
449     {
450         pthread_cond_signal(&use);
451     }
452     pthread_mutex_unlock(&mutex); // Unlock mutex
453     pthread_detach(pthread_self()); // Release resources
454 
455 }
456 
457 /**
458  * Validate command line parameters
459  * @param argc number of parameters
460  * @param argv command line parameters
461  */
462 void validateUse(int argc, char* argv[])
463 {
464     // Ensure correct number of command line parameters
465     if (argc != 3)
466     {
467         printf("Ensure there are the correct number of parameters\n");
468         exit(1);
469     }
470 
471     // Ensure maxDelay is positive
472     if ( atoi(argv[2]) < 0)
473     {
474         printf("The maxDelay must be non-negative\n");
```

```
475             exit(1);
476         }
477 }
```

5.5 Threads: mssv.h

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <fcntl.h>
#include <semaphore.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <time.h>
#include <pthread.h>

#define NINE 9
#define SUB 3
#define NUMTHREADS 11
#define FALSE 0
#define TRUE !FALSE

typedef enum {ROW, COL, SUB_GRID} Region_Type;

typedef struct
{
    Region_Type type;
    int position;
    pthread_t tid;
    int count;
    int valid;
    int numbers[NINE];

} Region;


void readFile(char* inputFile, int rows, int cols, int***buffer);
void writeFile(Region* region, char* format);
void resetArray(int numbers[]);
int checkValid(int numbers[]);
void parentManager(void);
void* childManager(void* args );
void initMemory( int*** buff1, int** buff2, int** counter, Region** regions);
void mapMemory(int* buff1FD, int* buff2FD, int* counterFD, int* semFD,
                    int* regionFD, int* resFD, int (**buff1Ptr)[NINE][NINE],
                        int (**buff2Ptr), int** countPtr, sem_t** semaphores,
                            Region** region, int** resourceCount);
void validateUse(int argc, char* argv[]);
void cleanMemory(void);
```

5.6 Threads: Makefile

```
# Makefile For Sudoku Solution Validator
# COMP2006 Assignment
# Last Modified: 12/04/17
# Jordan Yeo - 17727626

# MAKE VARIABLES
EXEC1 = mssv
OBJ1 = mssv.o
CFLAGS = -std=c99 -g -pthread -D _XOPEN_SOURCE=500 -lrt
CC = gcc
INPUT = ../testFiles/specTest.txt
DELAY = 10


# RULES + DEPENDENCIES
$(EXEC1) : $(OBJ1)
  $(CC) $(OBJ1) -o $(EXEC1) $(CFLAGS)

mssv.o : mssv.c mssv.h #fileIO.h
  $(CC) -c mssv.c $(CFLAGS)

#fileIO.o : fileIO.c fileIO.h
# $(CC) -c fileIO.c $(CFLAGS)

clean:
  rm -f $(EXEC1) $(OBJ1) logfile

run:
  ./$(EXEC1) $(INPUT) $(DELAY)
```